

3D GameStudio

Workshop Particles



**for A5 Engine 5.12
by Wolfgang Knecht March 2002**

The latest news, demos, updates and tools, as well as the Users' Magazine, the Users' Forum and the annual Contest are available at the GameStudio main page <http://www.3dgamestudio.com>.

Contents

Foreword.....	2
How to work with this workshop.....	3
Preparation.....	3
Let's start.....	4
Smoke.....	4
How to use smoke.....	7
Fire.....	9
How to use fire.....	12
Rain.....	13
How to use rain.....	15
Spell.....	17
How to use spell.....	20
Some hints for you at the end.....	21

Foreword

Dear Reader,

Welcome to the Particle Workshop! This Workshop should help you, making your own particle effects. The workshop is based on the new particle system which became available with the recent (5.12) update.

What are particles?

Particles are small 2D bitmaps which appear in thousands. They are used for effects like smoke, fire, tornados, rain and all the other cool effects you can imagine. You may think, you could also use sprites instead of these particles. In some cases this is true, but particles are much faster than sprites. Another difference between sprites and particles is, that particles have no collision detection.

Like the other workshops before, this workshop is meant to complement the rest of the documentation, not to replace it. I assume that you know the basics of **WDL** and **WED**.

I hope you like this workshop and learn something through it. Questions or comments can be sent to millennium-arts@gmx.at and I will try to help you.

Please excuse spelling and grammar mistakes. English isn't my native language.

Wolfgang "Knex | MA" Knecht

How to work with this workshop

In connection with particles there are a lot of features, which aren't implemented in each edition (e.g. alpha, flare, bright, ...). So I decided to make a difference between the code for lower and higher editions.

If a code snippet uses feature of higher editions, I will label it that way: 

Alternative code snippets for lower editions will be labelled that way: 

If the code snip is for all editions there will be a label like this: 

A label is always valid until the next label.

e.g.:

If you own the standard edition just follow  and .

If you own the professional edition follow  and .

This goes for indented lines only. Everything else is valid in general.

The whole code for lower editions is available in `pfxlow.wdl`. The whole code for higher editions is in `pfxhigh.wdl`.

Preparation

Get the latest update

Make sure you have the latest update of 3D Gamestudio. This is very important because starting with version **5.12** there is a new particle system in use so nearly every command relating to particles needs at least version **5.12** or higher.

Prepare your workspace

Create a folder called "Particle Workshop" in your GStudio folder. This folder will contain all files, you will need for this workshop.

Unzip the content of `PARTICLEFX.ZIP` into this folder. This folder should now contain the following files:

```
fire.pcx
fire.wmb
particle.wmp
pfxhigh.wdl
pfxlow.wdl
rain.pcx
smoke.pcx
smoke.wmb
spell.pcx
warlock.mdl
```

Create a level

In this workshop I will not explain, how to create a level step by step, because level architecture isn't really important for our particle effects. The only thing you should pay attention to, is to create something like a roof in your level because of the rain effect (we will talk about this effect later). You

should also have a player entity in your level with any movement action (e.g. `player_walk`). I think the best would be, if you use `particle.wmp` which comes with this workshop. Create a new WDL file called `particle.wdl` for this level (**File/Map Properties/New**).



Start with any level

Let's start

First of all we will create a simple smoke effect. Than we will talk about how a fire comes about. Another interesting effect will be a rain effect. After these effects we will create a nice spell.

We want to design this code so we can add it to several different projects with minimal amount of modification. To this end we are going to make a separate WDL-file which can easily be included in any of our projects.

We will start by creating a new plain text file called `pfx.wdl`. You can create this file in your project folder or you can create a new folder that you will use to store your own user defined template files. The first thing we are going to add is a comment block at the top of our code so when we are looking at this code months later we will know what the functions in this file do.

Smoke



```
////////////////////////////////////
// File: pfx.wdl
// WDL code fire, smoke, rain and spell - effects
////////////////////////////////////
```

Smoke

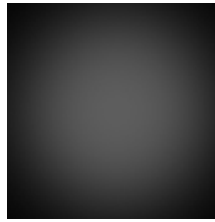
```
/*
 * Smoke effect *
 */
```



```
bmap smoke_parmap=<smoke.pcx>;
```

First we have to define a bitmap, which every smoke particle will get:

smoke.pcx



I'm not a good 2D artist, so I only use very simple bitmaps for my effects. If you want, you can make better bitmaps for better results.

```
function smoke_property();      // prototype
```



```
function smoke_effect();        // prototype
```

Then we have to define the functions, we will need for our smoke effect.

```
function smoke_effect() {
```

`Smoke_effect()` will be the function which sets the start values for each particle. In this function we will set properties like velocity, colors, transparency and so on.

```
my.vel_x=0;
my.vel_y=0;
my.vel_z=random(3);
```

With `my.vel_x`, `my.vel_y` and `my.vel_z` we set the velocity of the particles. Normally smoke has the characteristic to move upward along the Z-Axis. So we give `my.vel_z` a random value between 0 and 3. If you want, you can change this value to make the smoke moving upwards faster or slower. You may think, we have to multiply `my.vel_z` with time. We don't have to, `vel_x`, `vel_y` and `vel_z` are automatically relative to time.

```
my.move=on;
```

`my.vel_x`, `my.vel_y` and `my.vel_z` wouldn't have any effect if `my.move` is set off.

```
my.size=15;
```

`my.size` sets like the name says the size of the particles. You can also experiment with this value a bit. The default value of `my.size` is 4 but smoke doesn't look good, if you see small particles. So a larger value like 15 seems to be better.



```
my.bmap=smoke_pmap;
my.flare=on;
my.alpha=10;
```

You remember the `smoke_pmap`, we have defined before? Now it gets used. With `my.bmap` we assign `smoke_pmap` to the particle.

`My.flare` sets a flare effect to the bitmap of the particle. That means darker areas are more transparency than brighter ones. With `my.alpha` we are able to set the intensity of the transparency. If `my.alpha` is set to 100, the particle isn't transparent generally. If we set it to 0 the particle is totally transparent and we can't see it any more.



```
my.lifespan=64;
```

`My.lifespan` sets the lifetime of a particle in ticks. So the lifetime of a smoke particle will be 4 seconds (64/16 seconds). Default value of `my.lifespan` is 80 (5 seconds). `My.lifespan` counts continuously down. If it is 0 the particle will get removed. If you want to have each smoke particle for a longer or shorter time just change this value.

```
my.red=128;
my.green=128;
my.blue=128;
my.transparent=on;
```

`My.red`, `my.green` and `my.blue` set the RGB - colors of the particle. If all values are 128 the color of the particle will be grey like smoke should be.

```
my.function=NULL;
}
```

After setting the start values, we don't need anything else, so we set the function for the particle to NULL. The particles will live until `my.lifespan` reaches 0.



```
my.function=smoke_property;
}
```

Now that we've initialized the particle, we assign a new function to it which determines how the particle shall behave in its further life. Now we have to code things which should change with every repeat of the function `smoke_property()`.

```
function smoke_property() {
    my.alpha-=0.1*time;
    if (my.alpha<=0) {
        my.lifespan=0;
    }
}
```

We reduce `my.alpha` with every repeat of the function to fade out the smoke smoothly. Don't forget the factor `time` to avoid different effects on different computers! If the particle is faded out totally, `my.lifespan` is set to 0 and as a result the particle gets removed.

OK, our function for the particle is done. Now we need an other function which creates all the particles. The action smoke will be added to any entity.



```
action smoke {
    while (1) {
        temp.x=my.x+random(32)-16;
        temp.y=my.y+random(32)-16;
```

```
temp.z=my.z;
```

I assume you know what `while (1)` means. If not, it is needed for the endless loop (smoke rises up all the time).

We use the `temp` array for the start position of each smoke particle. The start position will be somewhere around the `my`-entity with a maximum distance of +/- 16 quants. If you want to change the distance in the `x` - or `y` - axis, just change the values 32 and 16 like you want (if the distance should be symmetrical, the first value has to be two times the second one.)

```
        effect(smoke_effect,5*time,temp,temp);  
        wait 1;  
    }  
}
```

With `effect` we create the particles. In the old particle system we have used the `emit` instruction for the same.

The first parameter (`smoke_effect`) assigns the function to each particle. Normally the particle function always needs to be before the `effect` instruction. Otherwise `effect` wouldn't find the function and an error would appear. But we had made prototypes of our functions at the beginning of the smoke effect, so it would work, also if the particle function is after the `effect` instruction.

The second parameter (`5*time`) determines the number of particles to be created. In that case `5*time` (every 1/16 second there will be created 5 new particle) is enough. If you want thicker smoke, you have to increase this value.

The third parameter (`temp`) sets the start position of the particles. These are the `temp.x`, `temp.y` and `temp.z` just above.

The last parameter (`temp`) sets the start velocity. In our effect we don't need this start property but have to hand over a parameter (otherwise: error) so we take the `temp` variable. It doesn't matter, because like you remember we have assigned another start velocity in the `smoke_effect` function.

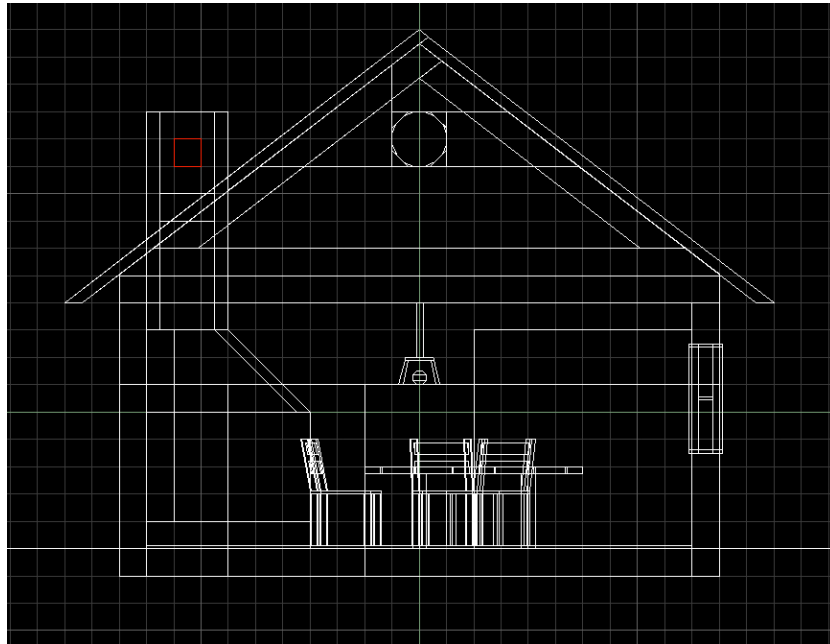
`Wait 1` ensures that there will be a break for other functions.

How to use smoke

The use of the smoke effect is very easy. Just `include pfx.wdl` below the other includes in the **WDL** file of your game.

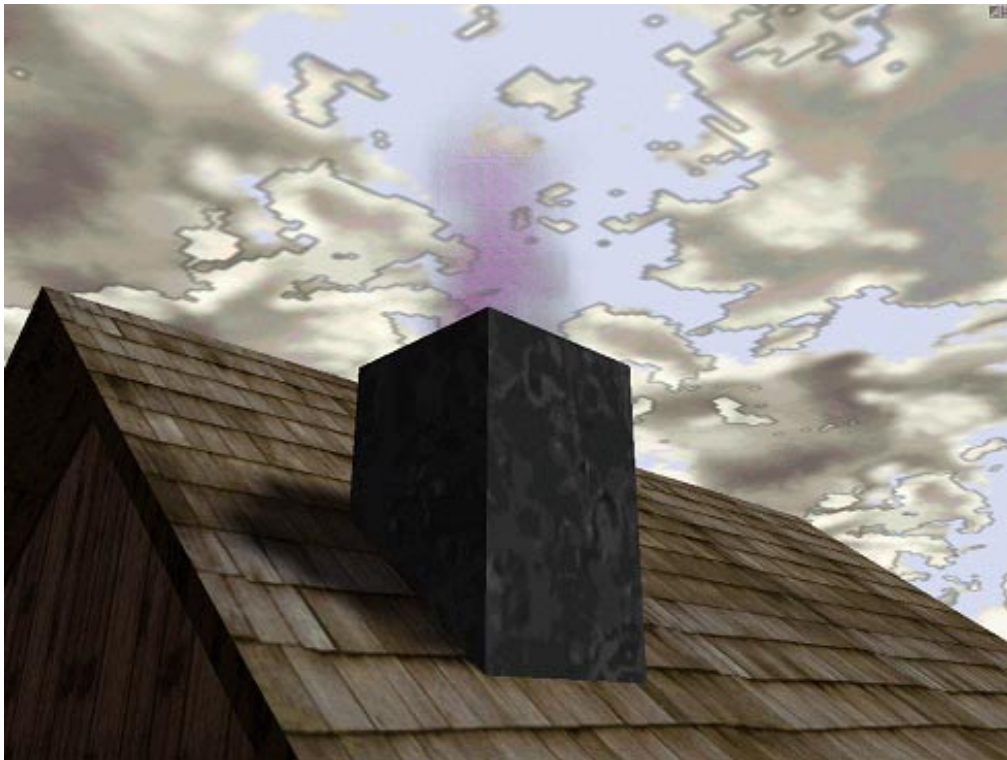
```
include <pfx.wdl>;
```

Then add an entity to your level where the smoke should be. As an entity you can use `smoke.wmb` for example.



including a smoke entity

Set the entity invisible and add the action smoke to it. Build and Run your level. That was it.



A smoking chimney

Fire

If you own a low edition of 3DGS, fire doesn't look very good, with particles. Sprites would be better. But for those who want to try it, there is an alternative code available, of course.

```

/*****
* Fire effect *
*****/

```



```
bmap fire_pmap=<fire.pcx>;
```

At first like for the smoke effect we have to define a bitmap for the fire effect:

fire.pcx



```

function fire_effect();
function fire_property();

```

And then we have to define the used functions.

For the fire we have to know a little bit algebra (if you aren't good at algebra – don't worry, I had to ask my older brother too :)).

```

function fire_effect() {
my.x+=random (30*(sin(total_ticks*200) * cos(total_ticks*50)));

my.y+=random (30*(sin(total_ticks*200) * sin(total_ticks*50)));

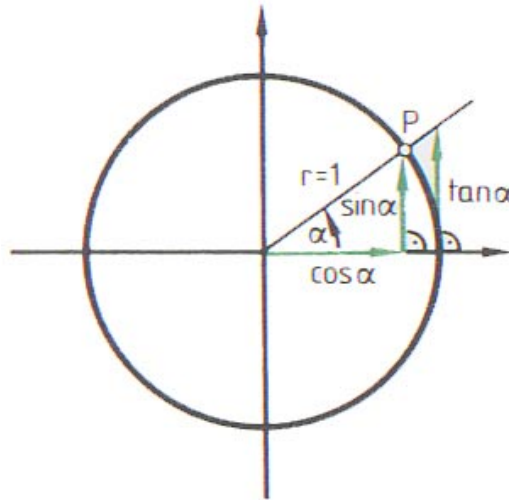
```

It isn't as complicated as it seems. Let's analyse it:

my.x, my.y and my.z is the start position (in the center of the fire) of each particle. But fire isn't only at one point, so we have to calculate random coordinates around the center of the fire. If you know algebra, you know that the coordinates of a simple circle are calculated like that:

$$x = \text{radius} * \sin(\text{angle})$$

$$y = \text{radius} * \cos(\text{angle})$$



In our case, angle is `total_ticks` multiplied by 50. `Total_ticks` is the number of ticks which have passed since we have started our game. So `total_ticks` counts up continuously like we need it. 50 is to speed it up.

Now we need a radius, which alternately becomes bigger and smaller. For something like that, `sin` is just perfect. `Sin` is always a number between -1 and 1. So our radius looks like that:

```
radius = 30*sin(total_ticks*200)
```

So the radius always will be between -30 and 30.

If we replace the function for coordinates of the circle above with the function we just found, the coordinates for a point, which moves on a spiral are:

```
x = 30*sin(total_ticks*200) * sin(total_ticks*50)
y = 30*sin(total_ticks*200) * cos(total_ticks*50)
```

At the end we just have to add a random.

```
my.size=15;
my.vel_x=0;
my.vel_y=0;
my.vel_z=2+random(2);
my.move=on;
```

We have to set the values of parameters, we already know from the smoke effect before.



```
my.bmap=fire_pmap;
my.flare=on;
my.bright=on;
my.alpha=25;
```

We already know almost all of the parameters above. `My.bright` is new. `My.bright` is very useful for fire since it gives the particle something like a glow effect. If there are some particles overlapping each other, they become brighter.

Watch at fire, you will notice that it is brighter in the middle than at it's edges. Exactly for that effect we need `my.bright`.



```
my.lifespan=16;
my.red=255;
my.green=255;
my.blue=0;
my.transparent=on;
```

We already know the results of these parameters. The lifetime of the particles will be 1 second (16/16 seconds). Because of the `my.red`, `my.green` and `my.blue` settings, the color will be yellow. Like smoke, fire is transparent too.



```
my.function=fire_property;
}
```

'Just-born' function is done.

```
function fire_property() {
```



```
my.alpha-=1*time;
if (my.alpha<=0) {
my.lifespan=0;
}
}
```

It's almost the same like with the smoke effect. The only difference is, that it will fade out a little faster.



```
my.red-=5*time;
my.green-=15*time;
}
```

If you watch at fire, you will notice, that it is more intensive in the center. The colors change. In the center it is yellow (sometimes almost white) and outwardly it becomes a darker red.

So we adjust the color of the particles with every repeat of the function. You can change this values (5 and 15) a bit if you want. I tried some other values and think that it looks mostly like real fire with the settings above.

You may think, we have to check, if `my.red` and `my.green` have a value between 0 and 255. It wouldn't be wrong, to do so, but our engine checks that automatically.



OK, the first part of our fire effect is done. The second part isn't too hard.

```
action fire {
  While (1) {
```

We all know what the instruction above does.

Like in the smoke effect we use `temp` for the start vector. Of course we could also use another vector variable, but in our case, `temp` does the job. Such a function looks like that:

```
temp.x=my.x;  
temp.y=my.y;  
temp.z=my.z;
```

We set `temp.x`, `temp.y` and `temp.z` to the start position.

```
    effect(fire_effect,20*time,temp,temp);  
    wait 1;  
}
```

The instruction `effect` isn't new to us. We now create every 1/16 seconds 20 particles. If you want to make the fire more or less intensive, just change this value.

How to use fire

The use of fire is the same like the use of smoke. Just include `pfx.wdl` in the **WDL** file of your game, add an entity where the fire should be and assign the action `fire` to it. As an entity you can use wood (e.g. `fire.wmb`) or whatever you want.



by the fireside

Rain

```

/*****
* Rain effect *
*****/

```

This rain effect you can easily adjust to fit your particular needs. You will be able to determine the intensity of the rain and the direction/velocity of the wind.



```
bmap rain_parmap=<rain.pcx>;
```

rain.pcx



```

function rain_effect();
function rain_property();

```

```
var raining=0;
```

With that rain effect, we will be able to switch the rain on and off, so we need a variable to check, if it is still raining. If `raining` is 0 it doesn't rain. If `raining` is 1 it's raining.

```

function rain_effect() {
my.vel_z=(-12.5-random(5));
my.x=camera.x+random(1500)-750;
my.y=camera.y+random(1500)-750;

```

Now we set the start position of the particle. We don't want to create particles all over the level. This would make the performance to go down. So we only create rain around the position of the camera (+/- 750 quants in the x and y axis relative to the camera position). The player won't realize that. If you want to, you can adjust the range of the rain like you need it.

```

my.skill_x=my.x;
my.skill_y=my.y;
my.skill_z=player.z+player.min_z;

```

We set `my.skill_x`, `my.skill_y` and `my.skill_z` at the position where the rain particle at least should disappear, if it doesn't hit another block before. We will need this skills for a trace function soon.

```

my.z=camera.z+500;
my.move=on;

```

500 quants above our head it will begin to rain. If your level contains a very high building, you might need to start the rain higher than that. Because if your building is higher than the

rain's starting point, it will rain inside. But for usual Levels 500 quants are more than enough.

```
my.x-=33*my.vel_x;
my.y-=33*my.vel_y;
```

If the wind is very strong, it may happen, that it doesn't rain at the amera's position but beside. So we need to correct this by setting the start position relating to the wind velocity.



Rain misses the player



That's much better

```
trace_mode = ignore_me;
trace(my.x,my.skill_x);
my.skill_z=target.z;
```

This is the `trace` function, I was talking about above. Here we check, if there is something between the start position of each particle and the ground. `My.skill_z` gets the `z` coordinate where the particle will hit something.



```
my.bmap=rain_parnap;
my.flare=on;
```

You should already know, what it means. If not, look at the first effect (smoke).



```
my.size=2;
my.red=0;
my.green=0;
my.blue=128;
```

Size and color are set.



```
my.function=rain_property;
}
```

Another 'just-born' function is done.

```
function rain_property() {
  if (my.z<=my.skill_z) {
    my.lifespan=0;
  }
}
```

```
}
```

Particle gets removed, if it hit something. So it can not rain inside a house, but you can see the rain trough windows.

OK, we have finished the particle function, let's go on with the function, which creates the particles.

```
function rain(vel_x,vel_y,intensity) {
```

If the rain function is called, it gets three parameters. With the first two you can set the velocity of the wind, with the third one, you can adjust the intensity of the rain.

```
    if (intensity==0) {intensity=20;}
```

If `intensity` isn't set, it wouldn't rain at all. So we set it to 20, a reasonable value.

```
    if (raining==1) {
        raining=0;
    } else {
        raining=1;
    }
}
```

Checks if it is raining, or not. If it is still raining, it stops, otherwise it starts.

```
    while (raining==1) {
        temp.x=vel_x;
        temp.y=vel_y;
```

If `raining` is 1, it should rain, so we set `temp.x` and `temp.y` to the direction/velocity of the wind. We need that to hand it over to the particle function (you remember the instruction in the 'just-born' function?)

```
        effect(rain_effect, intensity*time, temp, temp);
        wait 1;
    }
}
```

`effect` creates our particles. Like you know, the first parameter sets the particle function, the second the number of particles to be created, the third their start position (we don't need that parameter in our case, because we set the start position within the particle function) and the last one sets the start velocity (here for the wind).

How to use rain

Just include `pfx.wdl` in the **WDL** file of your game (`particle.wdl`) and call the function `rain` if you want to start the rain.

```
rain();
```

Simple rain, which comes down straight.

```
rain(20,10,50);
```

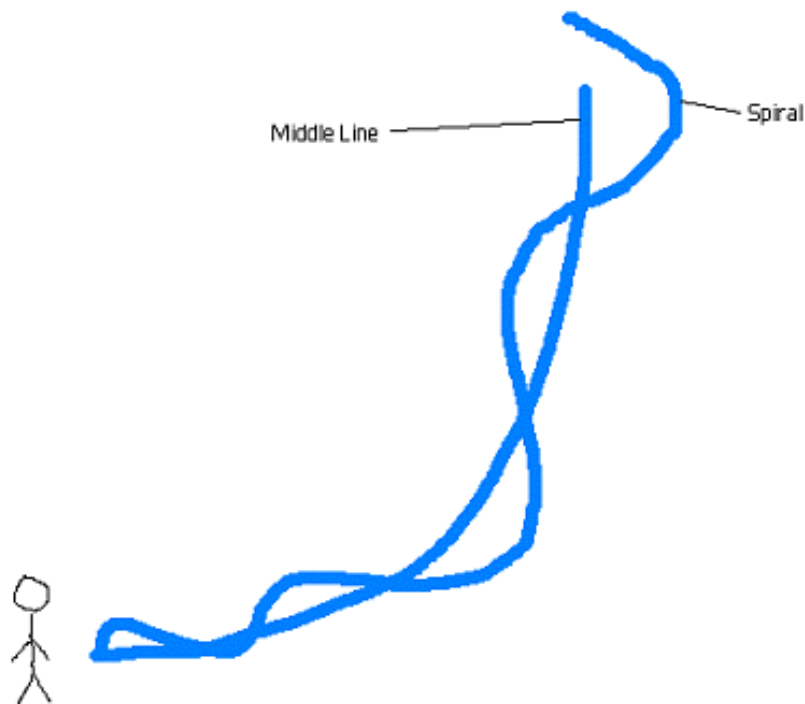



Heavy rain from the side.

If you want to stop the rain, just call `rain()` again.

Spell

First of all, if we want to create a spell effect, we have to think about, what it should look like.



Haven't I told, that I'm a horrible 2D artist?

The effect should start at the player's position and move upwards to the sky. There should be a central line and around this we want to be a spiral. The particles should fade out and fall down to the ground. The radius of our spiral should increase continuously. It's confusing, isn't it? If you don't understand, what I mean, just go on, and you will understand.

```

/*****
* Spell effect *
*****/

var spell_angle=0;
var spell_radius=0;
var spell_time=0;
var spell_existing=0;

```

We need some variables. `spell_angle` determines the direction of the spell. If the player looks to the south, the spell should appear in the south. Does the player look to the east, we want the spell to turn up there.

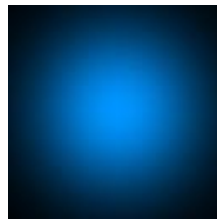
`spell_radius` sets the radius of the spiral around the center line of the spell. `spell_radius` increases continuously.

`spell_time` contains the age of the spell effect in ticks. The effect takes 50 ticks. And we need `spell_existing` to check, if there already is a spell effect is existing at the moment. If `spell_existing` is 1, another spell is existing and we shouldn't be able to start our spell until `spell_existing` reaches 0.



```
bmap spell_parmap=<spell.pcx>;
```

spell.pcx



```

function spell_effect();
function spell_property();

function spell_effect() {
    my.size=10;
    my.vel_x=0;
    my.vel_y=0;
    my.vel_z=0;
    my.move=on;

```



```

    my.bmap=spell_parmap;
    my.flare=on;
    my.bright=on;
    my.alpha=25;

```



```
my.lifespan=16;
my.red=0;
my.green=150;
my.blue=255;
my.transparent=on;
```



```
my.function=spell_property;
}
```

We also have to set the start parameters. If you aren't sure about an instruction above, just look at the previous effects.

```
function spell_property() {
```



```
my.alpha-=1*time;
if (my.alpha<=0) {
    my.lifespan=0;
}
```

The particle fades out and gets removed, as soon as it isn't visible any more.



```
my.gravity=2;
}
```

`My.gravity` sets the gravity (the same as `my.vel_z * (-1)`). Every particle will move downward continuously during it's whole life.

That wasn't too difficult. Now here comes the harder part.

```
function spell() {
    if (spell_existing==0) {
        spell_existing=1;
        spell_time=0;
        spell_angle=0;
        spell_radius=0;
        spell_angle=player.pan;
```

First we check, if an other spell is existing at the moment. If not (`spell_existing=0`), we can create a new one. Then we set our variables to the start values. We have to know, what direction the player faces, to make sure, the spell starts in that direction.

```
while (spell_time<50) {
    temp.x=player.x;
    temp.y=player.y;
    temp.z=player.z;
```

The lifetime of the spell will be 50 ticks (about 3 seconds). Start position is at the position of the player.

```
temp.x+=sin(90 - spell_angle) * cos (min
    (360,spell_time*2+270))*500;
temp.y+=cos(90 - spell_angle) * cos (min
    (360,spell_time*2+270))*500;
```

```
temp.z+=sin (min (360, spell_time * 2
+270))*500+500;
```

The vector `temp` is now a position at a quarter of a circle relative to the lifetime of the spell -> `spell_time`.

```
effect(spell_effect,1*time,temp, temp);
```

Now we create particles at that points along the curve. That will look like that:



This forms the center line of the spell. Surrounding this line, we will create particles along a spiral.

```
temp.x+=sin(- spell_angle) * cos
(spell_time*50)*spell_radius;
temp.y+=cos (- spell_angle) * cos
(spell_time*50)*spell_radius;

temp.z+=cos(- spell_time * 2) * sin
(spell_time*50)*spell_radius;
temp.x+=sin(90-spell_angle)*sin(-spell_time
*2)*sin(spell_time*50)*spell_radius;
temp.y+=cos(90-spell_angle)*sin(-spell_time
*2)*sin(spell_time*50)*spell_radius;
```

We have to bear in mind, that the spiral should be perpendicular to the center line. So the calculation of the position of every single particle along the spiral turns out to be a little confusing. If you don't understand the sinus and cosinus instruction I would suggest to look into a math book. This instructions are sometimes very helpful while creating computer games. You can produce nice effects, with that knowledge.

```
effect(spell_effect,1*time, temp, temp);
```

`effect` creates the particles along the spiral.

```
        spell_radius+=1*time;  
        spell_time+=time;  
        wait 1;  
    }
```

Spell_radius and spell_time increase continuously.

```
        rain(0,0,0);  
        spell_existing=0;  
    }  
}
```

After the effect is over, we can set an action/function to define, what shall happen, if the spell effect is called. In that effect we call the `rain` function. So we can let it rain (or stop it, like you remember) with calling that spell.

The variable `spell_existing` is set to 0. That means that no spell is existing at the moment.

How to use spell

The use of the spell differs a little from the others. This time you need to include `pxf.wdl` AFTER the `movement.wdl` within your game's **WDL** (`particle.wdl`). This is because within our funktion we use the `player` synonym defined in `movement.wdl`. So if you use a movement action of your own, you need to implement a `player` synonym.

To assign a key to the spell function, just use the `key_set` instruction in the main function (or where ever you want to assign a new key).

e.g.:

```
key_set(key_for_str("S"),spell);
```

If you hit **[S]**, the spell starts.



working spell

Some hints for you at the end

- Never forget to multiply the values, which change continuously in your particle effect with time to avoid different results on different computers.
- Don't use a `wait/waitt` instruction in the particle function. The engine wouldn't like that.
- Use the fourth parameter of the effect instruction to hand over values to the particle function. When a particle gets born, `my.vel_x`, `my.vel_y` and `my.vel_z` have these values, which were handed over by the effect instruction. If you assign `my.vel_x` for example to `my.skill_x`, you can use this value for other things, not only for the velocity.